Theoretical Computer Science 83 (1991) 57-69
Elsevier

About primitive recursive algorithms

Loïc Colson

LIENS and INRIA, URA 1327 du CNRS, Ecole Normale Supérieure, 45 rue d'Ulm, 75230 Paris cedex 05, France

Abstract

Colson, L., About primitive recursive algorithms, Theoretical Computer Science 83 (1991) 57-69.

In the past few years, there has been a growing interest in the application of proof-theoretical methods to the design of functional programming languages [3, 11]. One approach relies on representation theorems [3, 8, 10], which show that a large class of general recursive functions can be encoded in a language where general recursion is replaced by primitive recursion with functions, functionals, ... as parameters [13]. These results are however purely *extensional* in nature: they state that a large class of *mathematical functions* is representable in a given system, but they say nothing about the efficiency of such a representation. Although the *intensional* aspect is of primary concern for computer science, very little seems to be known about this question. This paper is a beginning in the study of this problem. We take as a case study the following computational model: a primitive recursive function is seen as defining a rewriting system which is evaluated in call-by-name. In this setting, we give a non-trivial necessary condition for an algorithm to be representable. As an application, we can show that the function inf (which computes the minimum of two integers in unary representation) cannot be programmed in complexity $O(\inf(n, p))$. Our proof method uses some basic notions of denotational semantics.

Introduction

In this paper, we study the representation problem in a functional system from an intensional point of view. What can we say about *algorithms* (as opposed to graphs or extensional mathematical functions) representable in a given functional system?

For instance, it is known that the class of functions representable in Gödel system T [5] contains strictly the class of primitive recursive functions (for example, the Ackermann function is also definable in system T). However, this is of little use from a computational point of view, since non-primitive recursive functions are intractable.

It seems that the intensional version of the same problem is more interesting. By the use of denotational semantics we show that primitive recursive algorithms use really only one argument for large incomplete inputs. As an application of this, with a simple notion of complexity of programs (basically, the number of reduction

steps), it is shown that the function inf *cannot* be computed by a primitive recursive algorithm in complexity $O(\inf(n, p))$, with a lazy evaluation mechanism.

However, this function may be represented by such a program in a functional system where we allow functional parameters (as in Gödel's system T, or Martin-Löf's type theory [5, 11]). This paper is thus an attempt to give a theoretical justification of the use of functional parameters in programming languages.

First, we give a possible formalisation of the notion of primitive recursive functions, as combinators with a given number of arguments. Then, we recall the standard denotational meaning of these algorithms. This is used to show that such an algorithm, at a certain point of its computation, will only explore one of its arguments before having exhausted it. From this, we derive a method of proving that a particular (intensional) algorithm is not primitive recursive. We give an example which shows that this property becomes false if we use functional parameters.

1. Primitive recursive functions

The goal is to give a convenient formalisation of the notion of primitive recursive function. In our approach, we define first the notion of primitive recursive combinator of a given arity, intuitively a function waiting for its arguments, and then the notion of primitive recursive term, which is a primitive recursive combinator in a given environment.

1.1. Primitive recursive combinators

They are defined inductively:

- O is a primitive recursive combinator of arity 0.
- *Succ* is a primitive recursive combinator of arity 1.
- π_i^n is a primitive recursive combinator of arity n (with $1 \leqslant i \leqslant n$), which represents the ith n-ary projection.
- $S_m^n(c; c_1, \ldots, c_n)$ is a primitive recursive combinator of arity m, provided that c is a primitive recursive combinator of arity n and the c_i's are primitive recursive combinators of arity m (in the case $n = 0$ we write $S_m^0(c)$). This represents composition.
- $Rec(b, s)$ is a primitive recursive combinator of arity $n+1$, provided that b is a primitive recursive combinator of arity n, and s is a primitive recursive combinator of arity $n+2$. This represents primitive recursion with basic case b and induction step s.

As examples, π_1^1 is the identity, $S_2^0(O)$ is the binary null function. A primitive recursive combinator for addition is given by $\mathsf{add} = Rec(\pi_1^1, S_3^1(Succ; \pi_2^3))$, which corresponds intuitively to the rewriting system:

- $\mathsf{add}(0, y) = y$
- $\mathsf{add}(Succ(x), y) = Succ(\mathsf{add}(x, y))$.

1.2. Primitive recursive terms

They are defined inductively: if c is a primitive recursive combinator of arity n, and $t_1, \ldots, t_n$ are primitive recursive terms, then $c[t_1, \ldots, t_n]$ is a primitive recursive term.

2. Denotational semantics

For representing lazy integers, we use the domain **D** solution of the domain equation $\mathbf{D} \simeq \mathit{void} \oplus \mathbf{D}$ (where $\mathit{void} = \{\bot\}$ and $\oplus$ is the lifted sum) (fig. 1). For this domain, we have a constant $0 \in \mathbf{D}$ and a function $S : \mathbf{D} \to \mathbf{D}$. We can consider the subset $\mathbf{D}_0$ of "complete elements" of the form $S^k(0)$ and its complement $\mathbf{D}_1$ of "incomplete elements" of the form $S^k(\bot)$ or $S^\omega(\bot)$. Then, we can give by structural induction a semantics of a primitive recursive combinator of arity n as a continuous function from $\mathbf{D}^n$ to **D** (so that $\mathbf{D}_0^n$ is sent into $\mathbf{D}_0$).

- $[\![O]\!]$ is $0 \in \mathbf{D}$,
- $[\![\mathit{Succ}]\!]$ is $S \in \mathbf{D} \to \mathbf{D}$,
- $[\![\pi_i^n]\!]$ is the ith n-ary projection,
- $[\![S_m^n(c; c_1, \ldots, c_n)]\!]$ is $[\![c]\!] \circ ([\![c_1]\!], \ldots, [\![c_n]\!])$,
- $F = [\![\mathit{Rec}(b, s)]\!]$, of arity $n+1$, is defined on finite elements of $\mathbf{D}^{n+1}$ in the following way: $F(\bot, u_1, \ldots, u_n) = \bot$, $F(0, u_1, \ldots, u_n) = [\![b]\!](u_1, \ldots, u_n)$ and $F(S(u), u_1, \ldots, u_n) = [\![s]\!](u, F(u, u_1, \ldots, u_n), u_1, \ldots, u_n)$; it then extends uniquely to all elements by continuity.

For instance, we have $[\![\mathsf{add}]\!](0, 0) = 0$, $[\![\mathsf{add}]\!](\bot, 0) = \bot$, $[\![\mathsf{add}]\!](S(0), S(\bot)) = S^2(\bot)$ and $[\![\mathsf{add}]\!](S^\omega(\bot), S^r(\bot)) = S^\omega(\bot)$.

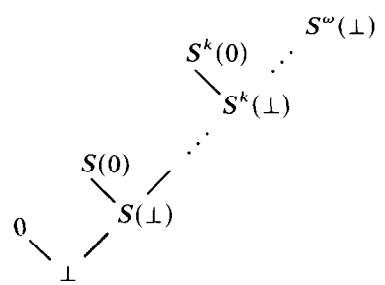

Fig. 1. The domain **D** of lazy natural numbers.

3. A denotational result

Intuitively, an element of the domain **D** represents abstractly some information about the computation. Thus $[\![c]\!](v_1, \ldots, v_n)$, for c a primitive recursive combinator

of arity n, and $(v_1, \ldots, v_n) \in \mathbf{D}^n$ a finite element, will represent the computation of the primitive recursive term $c[t_1, \ldots, t_n]$, if v_i is the representation of the computation of t_i. There are two cases:

- $[\![c]\!](v_1, \ldots, v_n)$ is an integer $S^k(0)$, that means that the computation terminates requiring at most the amount of information $(v_1, \ldots, v_n)$,
- $[\![c]\!](v_1, \ldots, v_n)$ is some "incomplete information" $S^k(\perp)$, in which case, the intuition is that the computation is stopped because of a lack of information about one of the arguments (and, as we shall see, only one).

Example 3.1. The computations represented by $[\![\mathsf{add}]\!](\perp, 0)$ and $[\![\mathsf{add}]\!](S(0), S(\perp))$ are stopped respectively by the first and second argument.

Our main result is then the following property for c a primitive recursive combinator of arity n: for any $v \in \mathbf{D}^n$, possibly infinite, such that $[\![c]\!](v) \in \mathbf{D}_1$ there exists finite $v_0 \leqslant v$, such that all computations $[\![c]\!](w)$, for w finite such that $v_0 \leqslant w \leqslant v$, are stopped by the *same* argument. In dynamic terms, this amounts to saying that from a certain point of its computation, such a primitive recursive algorithm visits only one of its arguments, before having exhausted it.

The definition and properties of the next section capture formally this intuitive notion of "the argument that stops the computation" for primitive recursive algorithms.

3.1. Sequentiality of primitive recursive algorithms

The proofs of this section are of a purely routine character and can be omitted by "grasshopper" readers.

Definition 3.2. For a primitive recursive combinator c of arity n we define, for $1 \leqslant i \leqslant n$, the subset $X_i(c)$ formed of finite elements $(v_1, \ldots, v_n)$ of $\mathbf{D}^n$, such that the computation represented by $[\![c]\!](v_1, \ldots, v_n)$ is stopped by the ith argument. In order to simplify the notation, we write (v, w) for $(v, w_1, \ldots, w_n)$ if $w = (w_1, \ldots, w_n)$. By induction over c, we define:

- $X_1(\mathit{Succ})$ is the set of finite elements of $\mathbf{D}_1$,
- $X_j(\pi_i^n)$ is $\emptyset$ if $i \neq j$, and $X_i(\pi_i^n)$ is $\{(v_1, \ldots, v_n) \text{ finite} \mid v_i \in \mathbf{D}_1\}$,
- $X_i(S_m^n(c; c_1, \ldots, c_n))$ is $\bigcup_{1 \leqslant j \leqslant n} \{v \in X_i(c_j) \mid ([\![c_1]\!](v), \ldots, [\![c_n]\!](v)) \in X_j(c)\}$,
- $X_i(\mathit{Rec}(b, s))$, for $2 \leqslant i \leqslant n+1$, is the union of the following sets:
  - the set of elements $(0, v)$ with $v \in X_{i-1}(b)$,
  - the set of elements $(S(u_0), u)$ such that $(u_0, [\![\mathit{Rec}(b, s)]\!](u_0, u), u) \in X_{i+1}(s)$, or $(u_0, [\![\mathit{Rec}(b, s)]\!](u_0, u), u) \in X_2(s)$ and $(u_0, u) \in X_i(\mathit{Rec}(b, s))$,
- $X_1(\mathit{Rec}(b, s))$ is the union of the following sets:
  - the set of elements of the form $(\perp, v)$, v finite,
  - the set of elements $(S(u_0), u)$ such that $(u_0, [\![\mathit{Rec}(b, s)]\!](u_0, u), u) \in X_1(s)$, or $(u_0, [\![\mathit{Rec}(b, s)]\!](u_0, u), u) \in X_2(s)$ and $(u_0, u) \in X_1(\mathit{Rec}(b, s))$.

For instance, $X_1(\mathsf{add})$ is the set of finite elements in $\mathbf{D}_1 \times \mathbf{D}$ and $X_2(\mathsf{add})$ is the set of finite elements in $\mathbf{D}_0 \times \mathbf{D}_1$.

From this definition, it follows that if $v \in X_i(c)$ then $[\![c]\!](v) \in \mathbf{D}_1$, and $v_i \in \mathbf{D}_1$.

Proposition 3.3. *The union over i of all $X_i(c)$, for a given primitive recursive combinator c of arity n, is a downward closed subset of $\mathbf{D}^n$, which coincides with the set of finite $v \in \mathbf{D}^n$ such that $[\![c]\!](v) \in \mathbf{D}_1$.*

Proof. By the previous remark, it is sufficient to show that, if $[\![c]\!](v) \in \mathbf{D}_1$, then there exists j such that $v \in X_j(c)$. By structural induction on c.

Case 1. $c = \pi_i^n$, the union of $X_j(c)$ is exactly $\{v \in \mathbf{D}^n \mid v \text{ finite} \wedge \pi_i(v) \in \mathbf{D}_1\}$.

Case 2. For $c = S_m^n(d; c_1, \ldots, c_n)$, if v finite and $[\![c]\!](v) \in \mathbf{D}_1$, then by the induction hypothesis, there is $1 \leq i \leq n$ such that $([\![c_1]\!](v), \ldots, [\![c_n]\!](v)) \in X_i(d)$. By the remark above, $[\![c_i]\!](v) \in \mathbf{D}_1$. Hence, by the induction hypothesis, there is at least one $1 \leq j \leq m$ such that $v \in X_j(c_i)$. Then $v \in X_j(S_m^n(d; c_1, \ldots, c_n))$.

Case 3. For $c = Rec(b, s)$, we reason by induction on the first component of the argument. There are three cases.

- The argument is of the form $(0, v)$. If $[\![c]\!](0, v) \in \mathbf{D}_1$, then $[\![b]\!](v) \in \mathbf{D}_1$, hence there is an i such that $v \in X_i(b)$, and so $(0, v) \in X_{i+1}(c)$.
- The argument is of the form $(\perp, v)$. We have $(\perp, v) \in X_1(c)$ for all finite v by definition.
- The argument is of the form $(S(v_0), v)$. If $[\![c]\!](S(v_0), v) \in \mathbf{D}_1$, then $[\![s]\!](v_0, [\![c]\!](v_0, v), v) \in \mathbf{D}_1$. Hence, by the induction hypothesis, we have an i such that $(v_0, [\![c]\!](v_0, v), v) \in X_i(s)$.
  - If $3 \leq i$, then we obtain $(S(v_0), v) \in X_{i-1}(c)$.
  - If $i = 1$, then $(S(v_0), v) \in X_1(c)$.
  - If $i = 2$, then, by the remark above, $[\![c]\!](v_0, v) \in \mathbf{D}_1$. Hence, by the induction hypothesis, there is a j such that $(v_0, v) \in X_j(c)$. And then, $(S(v_0), v) \in X_j(c)$. □

The next result states that "at most one argument stops a computation".

Proposition 3.4. *For any primitive recursive combinator c, the sets of finite elements $X_i(c)$, as i varies, are pairwise disjoint.*

Proof. By the structural induction on c.

Case 1. If $v \in X_i(S_m^n(c; c_1, \ldots, c_n))$ and $v \in X_j(S_m^n(c; c_1, \ldots, c_n))$, there exists $1 \leq k \leq n$ and $1 \leq l \leq n$ such that $v \in X_i(c_k)$, $([\![c_1]\!](v), \ldots, [\![c_n]\!](v)) \in X_k(c)$, $v \in X_j(c_l)$, and $([\![c_1]\!](v), \ldots, [\![c_n]\!](v)) \in X_l(c)$. By induction, we have that $k = l$. Hence $i = j$.

Case 2. If $v = (v_0, v)$ belongs to $X_i(Rec(b, s))$ and $X_j(Rec(b, s))$, by induction on v_0:

- If $v_0 = \perp$, then $i = j = 1$.
- If $v_0 = 0$ then $2 \leq i$, $2 \leq j$, $v \in X_{i-1}(b)$, and $v \in X_{j-1}(b)$. By the induction hypothesis, $i = j$.

- If v_0 is $S(u_0)$, by the induction hypothesis there exists a unique k such that $(u_0, [\![Rec(b, s)]\!](u_0, v), v) \in X_k(s)$. If $k=1$ then $i=j=1$. If $3 \leq k$ then $i=j=k-1$. If $k=2$ $(u_0, v) \in X_i(c)$ and $(u_0, v) \in X_j(c)$ so by the induction hypothesis on u_0 we have $i=j$. □

Thus, the $X_i(c)$ constitute a partition of the finite elements of $[\![c]\!]^{-1}(\mathbf{D}_1)$.

The next proposition shows that the semantics of our algorithms are sequential with the usual notion of sequentiality [15].

Proposition 3.5. *If $v \in X_i(c)$, $v \leq w$ and w finite, and $v_i = w_i$, then $w \in X_i(c)$, and $[\![c]\!](w) = [\![c]\!](v)$.*

Proof. By structural induction on c.

Case 1. If $v \in X_i(S_m^n(c; c_1, \ldots, c_n))$, there exists $1 \leq j \leq n$ such that $v \in X_i(c_j)$, and $([\![c_1]\!](v), \ldots, [\![c_n]\!](v)) \in X_j(c)$. By the induction hypothesis $w \in X_i(c_j)$, and $[\![c_j]\!](w) = [\![c_j]\!](v)$. Hence, $w \in X_i(S_m^n(c; c_1, \ldots, c_n))$, since $([\![c_1]\!](w), \ldots, [\![c_n]\!](w)) \in X_j(c)$. Furthermore, $[\![S_m^n(c; c_1, \ldots, c_n)]\!](w) = c([\![c_1]\!](w), \ldots, [\![c_n]\!](w)) = c([\![c_1]\!](v), \ldots, [\![c_n]\!](v)) = [\![S_m^n(c; c_1, \ldots, c_n)]\!](v)$.

Case 2. If $v = (v_0, v')$ belongs to $X_i(Rec(b, s))$.

- If $v_0 = 0$, then $w = (0, w')$ with $w' \in X_{i-1}(b)$ by induction hypothesis, hence $w \in X_i(Rec(b, s))$, and $[\![Rec(b, s)]\!](w) = [\![b]\!](w') = [\![b]\!](v') = [\![Rec(b, s)]\!](v)$.
- If $v_0 = \perp$, then $i=1$ and $w_0 = \perp$, $w \in X_i(Rec(b, s))$ and $[\![Rec(b, s)]\!](w) = \perp = [\![Rec(b, s)]\!](v)$.
- If $v_0 = S(v_0')$, then $w_0 = S(w_0')$, with $v_0' \leq w_0'$. There are two cases.
  - If $v \in X_i(Rec(b, s))$ with $2 \leq i$, we have then $(v_0', [\![Rec(b, s)]\!](v_0', v'), v') \in X_{i+1}(s)$, or $(v_0', [\![Rec(b, s)]\!](v_0', v'), v') \in X_2(s)$, and $(v_0', v') \in X_i(Rec(b, s))$. The first case is by induction hypothesis on the primitive recursive combinator. The second case is by induction on v_0, and then on the primitive recursive combinator.
  - If $v \in X_1(Rec(b, s))$, we have $v_0' = w_0'$ and $(v_0', [\![Rec(b, s)]\!](v_0', v), v) \in X_1(s)$, or $(v_0', [\![Rec(b, s)]\!](v_0', v), v) \in X_2(s)$, and $(v_0', v) \in X_1(Rec(b, s))$. The first case is by induction hypothesis on the primitive recursive combinator. The second case is by induction on v_0, and then on the primitive recursive combinator. □

For example, we have $(S^k(\perp), S^l(\perp)) \in X_1(\mathsf{add})$, thus if $S^l(\perp) \leq u$, then $(S^k(\perp), u) \in X_1(\mathsf{add})$ and $[\![\mathsf{add}]\!](S^k(\perp), S^l(\perp)) = [\![\mathsf{add}]\!](S^k(\perp), u) = S^k(\perp)$.

Let us define a *sequentiality index* of a pair (f, v), $f \in \mathbf{D}^n \to \mathbf{D}$, $v \in \mathbf{D}^n$, as an integer i, $1 \leq i \leq n$, such that if $v \leq w$ and $v_i = w_i$, then $f(v) = f(w)$. The last proposition shows that if $v \in X_i(c)$ then i is a sequentiality index of $([\![c]\!], v)$. Unfortunately this is not sufficient to characterize the sets $X_i(c)$ by means of $[\![c]\!]$: consider the example leftstrictand of [2], definable primitive recursively as follows (write it lsa for short):

$$\mathsf{lsa} = Rec(Rec(O, S_2^0(O)), S_3^1(Rec(O, S_2^0(S_0^1(Succ, O))); \pi_3^3))$$

or with more readable rewrite rules:

- $\mathsf{Isa}(O, O) = O$,
- $\mathsf{Isa}(O, Succ(y)) = O$,
- $\mathsf{Isa}(Succ(x), O) = O$,
- $\mathsf{Isa}(Succ(x), Succ(y)) = Succ(O)$.

The order of the rules means that we begin by a recursion over the first argument, next on the second one. We have that $[\![\mathsf{Isa}]\!](\perp, x) = [\![\mathsf{Isa}]\!](x, \perp) = [\![\mathsf{Isa}]\!](\perp, \perp) = \perp$ for all $x \in \mathbf{D}$, so the inputs 1 and 2 are both acceptable sequentiality indexes for $([\![\mathsf{Isa}]\!], (\perp, \perp))$. However it is clearly the *first* occurrence of $\perp$ which stops the computation. That is why we defined the sets $X_i(c)$ by induction on c and not by mean of its denotation $[\![c]\!]$.

3.2. A property of primitive recursive algorithms

The objective of this section is to generalise the notion of sequentiality index as defined above to infinite inputs. Before stating the theorem which will allow this extension, we need a technical lemma.

Lemma 3.6. *If* $(S^k(\perp), [\![Rec(b, s)]\!](S^k(\perp), v), v) \in X_2(s)$ *then* $(S^l(\perp), v) \in X_1(Rec(b, s))$, *for all* $l \leqslant k+1$.

Proof. By induction on l. If $l = 0$, the property results of the definition of $X_1(Rec(b, s))$. If it holds for l and $l+1 \leqslant k+1$, then $(S^l(\perp), [\![Rec(b, s)]\!](S^l(\perp), v), v) \in X_i(s)$ with $i = 1$ or $i = 2$. Indeed, since $(S^l(\perp), [\![Rec(b, s)]\!](S^l(\perp), v), v) \leqslant (S^k(\perp), [\![Rec(b, s)]\!](S^k(\perp), v), v)$, we have that $(S^l(\perp), [\![Rec(b, s)]\!](S^l(\perp), v), v)$ is in one of the X_i (by Proposition 3.3) and we must have $i \leqslant 2$ by Proposition 3.5.

- If $i = 1$, we have $(S^{l+1}(\perp), v) \in X_1(Rec(b, s))$ by definition of $X_1(Rec(b, s))$.
- If $i = 2$, we have $(S^{l+1}(\perp), v) \in X_1(Rec(b, s))$, since, by induction hypothesis $(S^l(\perp), v) \in X_1(Rec(b, s))$. □

We only need the case $l = k+1$.

We can now state our main theorem, which will allow us to extend $X_i(c)$ to infinite elements.

Theorem 3.7. *Let c be a primitive recursive combinator of arity n, and* $v \in \mathbf{D}^n$, *possibly infinite, such that* $[\![c]\!](v) \in \mathbf{D}_1$. *Then there exists one i,* $1 \leqslant i \leqslant n$, *and a finite* $v_0 \leqslant v$, *such that if w is finite and* $v_0 \leqslant w \leqslant v$, *then* $w \in X_i(c)$.

Proof. Notice that this theorem only really says something if v is infinite.

Case 1. For *Succ* and $v \in \mathbf{D}$ such that $S(v) \in \mathbf{D}_1$, we can take any finite $v_0 \leqslant v$ (note that $\mathbf{D}_1$ is downward closed).

Case 2. The case π_i^n is similar to the previous one.

Case 3. For $S^n_m(c; c_1, \ldots, c_n)$, and $v \in \mathbf{D}^m$, such that, if $u = (\llbracket c_1 \rrbracket(v), \ldots, \llbracket c_n \rrbracket(v)) \in \mathbf{D}^n$, we have $\llbracket c \rrbracket(u) \in \mathbf{D}_1$. By the induction hypothesis, there is a finite $u_0 \in \mathbf{D}^n$, and $1 \leqslant j \leqslant n$ such that, if $w \in \mathbf{D}^n$ is finite, and $u_0 \leqslant w \leqslant u$, then $w \in X_j(c)$. Hence we have $\pi_j(u) \in \mathbf{D}_1$, that is $\llbracket c_j \rrbracket(v) \in \mathbf{D}_1$. By the induction hypothesis and by continuity, there is a finite $v_0 \leqslant v$, and $1 \leqslant i \leqslant n$ such that if $v_0 \leqslant t \leqslant v$, t finite, we have both $u_0 \leqslant (\llbracket c_1 \rrbracket(t), \ldots, \llbracket c_n \rrbracket(t)) \leqslant u$, and $t \in X_i(c_j)$, hence the result.

Case 4. For $c = Rec(b, s)$, and $(v_0, v) \in \mathbf{D}^n$, such that $\llbracket Rec(b, s) \rrbracket(v_0, v) \in \mathbf{D}_1$. By case on v_0:

(a) If $v_0 \in \mathbf{D}_0$, then by induction on v_0:

- If $v_0 = 0$, we have $\llbracket b \rrbracket(v) \in \mathbf{D}_1$, and so, by the induction hypothesis we find $w \leqslant v$ finite, and i, such that $u \in X_i(b)$ for $w \leqslant u \leqslant v$, u finite. Then, $(0, w)$ is finite, and if $(0, w) \leqslant (0, u) \leqslant (0, v)$, we have $(0, u) \in X_{i+1}(c)$.
- If $v_0 = S(v_0')$, we have $\llbracket s \rrbracket(v_0', \llbracket Rec(b, s) \rrbracket(v_0', v), v) \in \mathbf{D}_1$. By the induction hypothesis, we have $(w_0, w_1, w) \leqslant (v_0', \llbracket Rec(b, s) \rrbracket(v_0', v), v)$ finite, and $1 \leqslant i \leqslant n+2$ such that if $(w_0, w_1, w) \leqslant t \leqslant (v_0', \llbracket Rec(b, s) \rrbracket(v_0', v), v)$, $t \in X_i(s)$. Since we have $i \geqslant 2$ (because $v_0 \in \mathbf{D}_0$), there are two cases.
  - If $3 \leqslant i \leqslant n+2$, the result follows from the continuity of $\llbracket Rec(b, s) \rrbracket$.
  - If $i = 2$, we have $\llbracket Rec(b, s) \rrbracket(v_0', v) \in \mathbf{D}_1$. By the induction hypothesis on v_0, and by continuity, we obtain (w_0', w') finite such that $(w_0, w) \leqslant (w_0', w')$ and j such that if $(w_0', w') \leqslant (t_0', t) \leqslant (v_0', v)$, then $w_1 \leqslant \llbracket Rec(b, s) \rrbracket(t_0', t)$, and $(t_0', t) \in X_j(Rec(b, s))$. For $(S(w_0'), w') \leqslant (t_0, t) \leqslant (S(v_0'), v))$, we have $t_0 = S(t_0')$ and $(w_0', w') \leqslant (t_0', t) \leqslant (v_0', v)$. Hence, $(t_0, t) \in X_j(Rec(b, s))$.

(b) If $v_0 \in \mathbf{D}_1$, there are two cases.

- If $v_0 = \perp$, then for any $w \leqslant v$ finite and $(\perp, w) \leqslant (t_0, t) \leqslant (v_0, v)$, we have $t_0 = \perp$ and hence $(t_0, t) \in X_1(Rec(b, s))$.
- If $v_0 = S(v_0')$, we have that $\llbracket s \rrbracket(v_0', \llbracket Rec(b, s) \rrbracket(v_0', v), v) \in \mathbf{D}_1$. By the induction hypothesis, we have $(w_0', w_1, w) \leqslant (v_0', \llbracket Rec(b, s) \rrbracket(v_0', v), v)$ finite, and $1 \leqslant i \leqslant n+2$ such that if $(w_0', w_1, w) \leqslant u \leqslant (v_0', \llbracket Rec(b, s) \rrbracket(v_0', v), v)$, $u \in X_i(s)$. By continuity, we can suppose that for $(w_0', w) \leqslant (t_0', t) \leqslant (v_0', v)$, $w_1 \leqslant \llbracket Rec(b, s) \rrbracket(t_0', t)$ (increase (w_0', w) if necessary).
  - If $3 \leqslant i \leqslant n+2$, we have that for (t_0, t) finite and $(S(w_0'), w) \leqslant (t_0, t) \leqslant (S(v_0'), v)$, $(t_0, t) \in X_{i-1}(c)$.
  - If $i = 1$, we have that for (t_0, t) finite and $(S(w_0'), w) \leqslant (t_0, t) \leqslant (S(v_0'), v)$, $(t_0, t) \in X_1(c)$.
  - If $i = 2$, we claim that for (t_0, t) finite, and $(S(w_0'), w) \leqslant (t_0, t) \leqslant (S(v_0'), v)$, $(t_0, t) \in X_1(c)$. Indeed, we have $t_0 = S(t_0')$, with $w_0' \leqslant t_0' \leqslant v_0'$, and so $(t_0', \llbracket Rec(b, s) \rrbracket(t_0', t), t) \in X_2(s)$. Hence, by the lemma (that we can apply, since $t_0' \in \mathbf{D}_1$) we obtain $(t_0, t) \in X_1(Rec(b, s))$. □

We can so extend the partition $X_i(c)$ of the finite elements of $\llbracket c \rrbracket^{-1}(\mathbf{D}_1)$ into a partition of the entire set $\llbracket c \rrbracket^{-1}(\mathbf{D}_1)$. We define $v \in X_i(c)$, for v possibly infinite, by $u \in X_i(c)$, for finite u sufficiently large, $u \leqslant v$. Intuitively, this means that the notion of "the argument that stops a computation" still has a meaning even for infinite

inputs. For instance, we have $(S^\omega(\perp), S^\omega(\perp)) \in X_1(\mathsf{add})$, which corresponds to the fact that add looks at its second argument only when it has exhausted its first argument.

4. Applications

4.1. A denotational example

We use the previous result to show that some algorithms are not primitive recursive.

Proposition 4.1. *There is no primitive recursive combinator c of arity* 2, *which satisfies* $[\![c]\!](S^n(\perp), S^p(\perp)) = S^{\inf(n,p)}(\perp)$.

Proof. Let c be a binary primitive recursive combinator. Let $u = [\![c]\!](S^\omega(\perp), S^\omega(\perp))$.

- If $u \in \mathbf{D}_0$ then for n, p large enough we have $[\![c]\!](S^n(\perp), S^p(\perp)) = u \neq S^{\inf(n,p)}(\perp)$ since $u \in \mathbf{D}_0$.
- If $u \in \mathbf{D}_1$ we have for instance, $(S^\omega(\perp), S^\omega(\perp)) \in X_1(c)$. That is, for n, p large enough, $(S^n(\perp), S^p(\perp)) \in X_1(c)$. Hence, by Proposition 3.5, we have $[\![c]\!](S^n(\perp), S^p(\perp)) = [\![c]\!](S^n(\perp), S^{p'}(\perp))$ for $p \leqslant p'$, and n, p big enough. In particular, for q big enough we have $[\![c]\!](S^{q+1}(\perp), S^q(\perp)) = [\![c]\!](S^{q+1}(\perp), S^{q+1}(\perp))$, so $[\![c]\!](S^{q+1}(\perp), S^q(\perp)) \neq S^{\inf(q+1,q)}(\perp) = S^q(\perp)$ or $[\![c]\!](S^{q+1}(\perp), S^{q+1}(\perp)) \neq S^{\inf(q+1,q+1)}(\perp) = S^{q+1}(\perp)$. Hence the result. □

Example 4.2. Consider the following rewriting system, which defines a modified version of Ackermann function:

- $A(O, y) = Succ(y)$,
- $A(Succ(x), O) = A(x, Succ(O))$,
- $A(Succ(x), Succ(y)) = Succ(A(x, A(Succ(x), y)))$.

Then, the (intuitive) denotation **A** of A verifies $\mathbf{A}(S^n(\perp), S^p(\perp)) = S^{\inf(n,p)}(\perp)$, and so is not primitive recursive. Note that this does not say that the *function* defined by the previous equations is not primitive recursive, but only that there is no primitive recursive algorithm that has the same denotation as the algorithm defined by this rewriting system (evaluated in call by name).

The next section gives a more interesting operational result.

4.2. A complexity corollary

First, we generalise our notion of primitive recursive terms. We simply add a dummy constant Ω.

We can then give a denotational semantics of the primitive recursive terms. We take $[\![\Omega]\!] = \perp$, and $[\![c[t_1, \ldots, t_n]]\!] = [\![c]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$.

We can give also an operational semantics. This is a deterministic rewriting system.

- $\pi_i^n[t_1, \ldots, t_n] \Rightarrow t_i$,
- $S_m^n(c; c_1, \ldots, c_n)[t_1, \ldots, t_m] \Rightarrow c[c_1[t_1, \ldots, t_m], \ldots, c_n[t_1, \ldots, t_m]]$,
- $Rec(b, s)[O, t_1, \ldots, t_n] \Rightarrow b[t_1, \ldots, t_n]$,
- $Rec(b, s)[Succ[u], t_1, \ldots, t_n] \Rightarrow s[u, Rec(b, s)[u, t_1, \ldots, t_n], t_1, \ldots, t_n]$,
- if $t \Rightarrow u$, $Rec(b, s)[t, t_1, \ldots, t_n] \Rightarrow Rec(b, s)[u, t_1, \ldots, t_n]$, provided the previous rule does not apply,
- if $t \Rightarrow u$, then $Succ[t] \Rightarrow Succ[u]$.

We say that a term is normal if, and only if, it cannot be reduced. The complexity $\mathsf{cost}(t)$ of a primitive recursive term is the length of the reduction sequence (t_i) of t (which is finite by usual normalisation argument).

We can note the following transfer principle: if (t_i) is the reduction sequence of t, then the reduction sequence of $t[\Omega/O]$ is at least $(t_i[\Omega/O])$, where $t[\Omega/O]$ denotes the term t where all occurrences of Ω have been replaced by O. Thus, $\mathsf{cost}(t) \leqslant \mathsf{cost}(t[\Omega/O])$.

Lemma 4.3. *Let $t = c[t_1, \ldots, t_n]$ be a primitive recursive term such that $[\![t]\!] = S^k(\bot)$, and $([\![t_1]\!], \ldots, [\![t_n]\!]) \in X_i(c)$. Then $[\![t_i]\!]$ is of the form $S^l(\bot)$ and $\mathsf{cost}(t_i) + l \leqslant \mathsf{cost}(t) + k$.*

Proof. By induction on c.

Case 1. For π_i^n, we have $t \Rightarrow t_i$ and $k = l$, hence the result.

Case 2. For $S_m^n(c; c_1, \ldots, c_n)$, we have $t \Rightarrow c[u_1, \ldots, u_n]$ with $u_j = c_j[t_1, \ldots, t_m]$. By definition, there exists $1 \leqslant j \leqslant n$ such that $([\![u_1]\!], \ldots, [\![u_n]\!]) \in X_j(c)$. Hence, by induction hypothesis, if $[\![u_j]\!] = S^p(\bot)$, $\mathsf{cost}(u_j) + p \leqslant \mathsf{cost}(t) + k$. Furthermore, $([\![t_1]\!], \ldots, [\![t_m]\!]) \in X_i(c_j)$. By induction hypothesis, $\mathsf{cost}(t_i) + l \leqslant \mathsf{cost}(u_j) + p$. Hence the result by transitivity.

Case 3. For $c = Rec(b, s)$, we can suppose that t_1 is of the form O or $Succ(u)$ or is normal of denotation $\bot$ (and we then prove the result by induction on $[\![t_1]\!]$).

- If $t_1 = O$, then $t \Rightarrow b[t_2, \ldots, t_n]$, hence the result by induction.
- If $t_1 = Succ(t_1')$, then $t \Rightarrow t'$ with $t' = s[t_1', c[t_1', t_2, \ldots, t_n], t_2, \ldots, t_n]$. Let us write $S^k(\bot) = [\![t]\!]$, $u_1' = [\![t_1']\!]$ and $u_i = [\![t_i]\!]$, and $u = [\![c]\!](u_1', u_2, \ldots, u_n)$. Then we have $(u_1', u, u_2, \ldots, u_n) \in X_j(c)$ with $1 \leqslant j \leqslant n+1$. If $3 \leqslant j$, the result follows by induction on the combinator. If $j = 1$, we have $u_1' = S^l(\bot)$ and $\mathsf{cost}(t_1') + l \leqslant \mathsf{cost}(t') + k$. But $\mathsf{cost}(t_1') = \mathsf{cost}(t_1)$ and $\mathsf{cost}(t') + 1 = \mathsf{cost}(t)$ hence $\mathsf{cost}(t_1) + (l+1) \leqslant \mathsf{cost}(t) + k$. If $j = 2$, the result follows by induction on $[\![t_1]\!]$.
- In the other cases, we have $[\![t_1]\!] = [\![t]\!] = \bot$, hence the result. □

Proposition 4.4. *There is no primitive recursive algorithm c which computes* inf, *and is of complexity* $\mathsf{O}(\mathsf{inf}(n, p))$.

Proof. Let us consider c a primitive recursive combinator of arity 2 that computes the function inf. Then, we must have $[\![c]\!](S^\omega(\bot), S^\omega(\bot)) \in \mathbf{D}_1$. We thus know by the

theorem, that for n, p big enough, we can suppose, without loss of generality, that $(S^n(\perp), S^p(\perp)) \in X_1(c)$. We must have $[\![c]\!](S^n(\perp), S^p(\perp)) \leqslant S^{\inf(n,p)}(\perp)$, since $[\![c]\!](S^n(\perp), S^p(\perp)) \leqslant [\![c]\!](S^n(0), S^p(0)) = S^{\inf(n,p)}(0)$ and $[\![c]\!](S^n(\perp), S^p(\perp)) \in \mathbf{D}_1$. Hence, by the lemma, we have $n \leqslant \mathsf{cost}(c[Succ^n(\Omega), Succ^p(\Omega)]) + \inf(n, p)$. By the transfer principle, we then obtain $n \leqslant \mathsf{cost}(c[Succ^n(O), Succ^p(O)]) + \inf(n, p)$. So choosing n large relative to p we see that c cannot have complexity $O(\inf(n, p))$. □

Notice that this is a very special kind of complexity result, not counting duplications of things or of computations to give an estimation like polynomial or exponential time, but rather a kind of I/O complexity result.

5. The importance of functionality

We show that we do have an algorithm for inf in complexity $O(\inf(n, p))$ if we allow functional parameters. We simply define F by the equations

- $F(O, y) = O$,
- $F(Succ(x), O) = O$,
- $F(Succ(x), Succ(y)) = Succ(F(x, y))$.

That is we take $F(O) = \lambda y.\ O$, $F(Succ(x)) = G(x, F(x))$, with G of type $Nat \to (Nat \to Nat) \to Nat \to Nat$, defined by $G(x, u)(O) = O$ and $G(x, u)(Succ(y)) = Succ(u(y))$. Note the use of the functional parameter u of type $Nat \to Nat$. This use of functional parameters was the original feature of Gödel system T. Such an algorithm F, when evaluated in call-by-name, computes $\inf(n, p)$ in $O(\inf(n, p))$ reduction steps. See [14, 4] for formalizations of system T, by means of combinators or lambda-calculus.

This little example shows that the use of functional parameters not only gives new functions (such as the Ackermann function), but may also yield better algorithms.

Krivine [9] shows that a particular pure λ-term computing the algorithm F is not correctly typable in second-order λ-calculus. However this does not show that no other term of this system computes the algorithm F. If we translate the previous term of system T in a second-order λ-term in the usual way, we obtain a term computing inf with the required property of visiting simultaneously its inputs, but with a wrong complexity: the built-in one-step predecessor of primitive recursion (crucially used in the previous definition) translates by the classical pairing technique into a *linear time* predecessor.

Conclusion and open problems

This paper deals with the following kind of question for functional systems: what can we say about the behaviour of algorithms, relative to a given evaluation mechanism, as opposed to the problem of representable functions (in term of graphs)? It is clear that even the class of primitive recursive functions is too large

for practical programming. However, our result shows that functionality is useful in order to get efficient algorithms for simple functions in the following framework: a program is a rewriting system that we get from the definition of a term in a functional system, evaluated in a lazy way. This work suggests first the following problem: to find a (sequential) algorithm, whose graph is a function provably total in Peano arithmetic, but which is not representable intensionally by a term of the system T (or more generally since the system T does not contain enough primitive data structures, the generalisation of T with any recursive type). Second, we proved that, for a primitive recursive algorithm, there is a "fixed sequentiality index" from a certain point in the computation: what is the behaviour of this sequentiality index for higher-order algorithms? Third, this paper is not completely constructive as it stands: we have to use a "principle of omniscience" when defining $[\![c]\!](S^\omega(\perp))$ as being either $S^k(\perp)$ for some k or $S^\omega(\perp)$. Looking for a constructive proof seems to raise interesting questions.

It is usual to classify functional systems via a hierarchy of fastly growing functions. For instance, if we use Grzegorczyk–Schwichtenberg hierarchy [3], f_ω corresponds to the system of primitive recursion (it is roughly the Ackermann function), and f_{ε_0} (well beyond Ackermann function) to the system T. This shows in a way the (theoretical) power of functionality. It would be interesting to illustrate this power by finding other concrete feasible algorithms with complex intensional behaviour.

Acknowledgement

The author is indebted to Thierry Coquand who suggested the main methodological tools of this paper. The initial motivation came from questions raised by J.L. Krivine and B. Maurey about the representation of inf in the system F (second order typed lambda-calculus). The author wants to thank Vincent Danos for stimulating discussions. Thanks too to Lars Ericson, Paris Kanellakis and Yves Lafont for useful suggestions and comments. Val Breazu-Tannen deserves special credit for a careful reading of this paper.

References

[1] W. Ackermann, Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen* **99** (1928) 1–36; English translation in J. van Heijenoort, *From Frege to Gödel* (Harvard University Press, 1981) 493–507.

[2] G. Berry and P.L. Curien, The kernel of the applicative language CDS: theory and practice, in: M. Nivat and J. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, 1985).

[3] S. Fortune, D. Leivant and M. O'Donnell, The expressiveness of simple and second-order type structures, *J. Assoc. Comput. Mach.*, **30** (1) (1983).

[4] J.Y. Girard, *Proof Theory and Logical Complexity, Vol.* 1, *Studies in Proof Theory* (Bibliopolis, Napoli, 1988).

[5] K. Gödel, Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, *Dialectica* **12** (1958) 280–287; English translation in: 'On a hitherto unexploited extension of the finitary standpoint, *J. Philos. Logic* **9** (1980).
[6] S.C. Kleene, *Introduction to Metamathematics* (North-Holland, Amsterdam, 1952).
[7] S.C. Kleene, Recursive functionals and quantifiers of finite types revisited II, in: *The Kleene Symposium* (North-Holland, Amsterdam, 1980).
[8] G. Kreisel, On the interpretation of nonfinitist proofs, Part I, II, *J. Symbolic Logic* **16, 17** (1952, 1953).
[9] J.L. Krivine, Un algorithme non typable dans le système *F*, in: *Comptes Rendus de l'Académie des Sciences*, Paris (1987).
[10] D. Leivant, Reasoning about functional programs and complexity classes associated with type disciplines, in: *24th IEEE Symp. on Foundations of Computer Science* (1983).
[11] P. Martin-Löf, Constructive mathematics and computer programming, in: *Logic, Methodology and Philosophy of Science* **6** (North-Holland, Amsterdam, 1980) 153–175.
[12] G. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (1976).
[13] J. Smith, Course-of-values recursion on lists in intuitionistic type theory, Unpublished notes, Göteborg University, 1981.
[14] W. W. Tait, Intensional interpretation of functionals of finite type I. *J. Symbolic Logic* **32** (1967) 198–212.
[15] J.E. Vuillemin, Proof techniques for recursive programs, Ph.D. thesis, Stanford, 1973.